Lecture 19

Using Classes Effectively

Announcements

Reading

- **Tuesday**: Chapter 18
- **Thursday** reading online

Assignments

- A4 due **tonight** at Midnight
  - 10 pts per day late
  - Consultants available tonight
- A5 & A6 posted tomorrow
  - See included *micro*-deadlines

Regrades

- Today is last day to request
  - Show it to me after class
  - I will verify if it is valid
- *Then* request regrade in CMS

- **Prelim, Nov 12th 7:30-9:00**
  - Material up to November 5
  - Recursion + Loops + Classes
- **S/U Students are exempt**
- **Conflict with Prelim time?**
  - Prelim 2 Conflict on CMS

Designing Types

From first day of class!

- **Type**: set of values and the operations on them
  - int: (**set**: integers; **ops**: +, –, *, /, …)
  - Time (**set**: times of day; **ops**: time span, before/after, …)
  - Worker (**set**: all possible workers; **ops**: hire,pay,promote,…)
  - Rectangle (**set**: all axis-aligned rectangles in 2D;
    ops: contains, intersect, …)
- To define a class, think of a *real type* you want to make
  - Python gives you the tools, but does not do it for you
  - Physically, any object can take on any value
  - Discipline is required to get what you want

Making a Class into a Type

1. Think about what values you want in the set
   - What are the attributes? What values can they have?
2. Think about what operations you want
   - This often influences the previous question

- To make (1) precise: write a *class invariant*
  - Statement we promise to keep true **after every method call**
- To make (2) precise: write *method specifications*
  - Statement of what method does/what it expects (preconditions)
- Write your code to make these statements true!

Planning out a Class

```
class Time(object):
    """Instances represent times of day.
    Instance Attributes:
        hour: hour of day [int in 0..23]
        min:  minute of hour [int in 0..59]"""

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""

    def increment(self, hours, mins):
        """Move this time <hours> hours
        and <mins> minutes into the future.
        Pre: hours is int >= 0; mins in 0..59"""

    def isPM(self):
        """Returns: this time is noon or later."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Time instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
class Rectangle(object):
    """Instances represent rectangular
    regions of the plane.
    Instance Attributes:
        t: y coordinate of top edge [float]
        l: x coordinate of left edge [float]
        b: y coordinate of bottom edge [float]
        r: x coordinate of right edge [float]
    For all Rectangles, l <= r and b <= t."""

    def __init__(self, t, l, b, r):
        """The rectangle [l, r] x [t, b]
        Pre: args are floats; l <= r; b <= t"""

    def area(self):
        """Return: area of the rectangle."""

    def intersection(self, other):
        """Return: new Rectangle describing
           intersection of self with other."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Rectangle instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Planning out a Class

```
class Hand(object):
    """Instances represent a hand in cards.
    Instance Attributes:
        cards: cards in the hand [list of card]
    This list is sorted according to the
    ordering defined by the Card class."""

    def __init__(self, deck, n):
        """Draw a hand of n cards.
        Pre: deck is a list of >= n cards"""

    def isFullHouse(self):
        """Return: True if this hand is a full
        house; False otherwise"""

    def discard(self, k):
        """Discard the k-th card."""
```

Class Invariant

States what attributes are present and what values they can have.

A statement that will always be true of any Rectangle instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Implementing a Class

- All that remains is to fill in the methods. (All?!)
- When implementing methods:
  1. Assume preconditions are true
  2. Assume class invariant is true to start
  3. Ensure method specification is fulfilled
  4. Ensure class invariant is true when done
- Later, when using the class:
  - When calling methods, ensure preconditions are true
  - If attributes are altered, ensure class invariant is true

Implementing an Initializer

```
def __init__(self, hour, min):
    """The time hour:min.
    Pre: hour in 0..23; min in 0..59"""
```

This is true to start

```
    self.hour = hour
    self.min = min
```

You put code here

```
Instance variables:
   hour: hour of day      [int in 0..23]
   min:   minute of hour [int in 0..59]
```

This should be true at the end

Implementing a Method

```
Instance variables:
    hour: hour of day      [int in 0..23]
    min:  minute of hour   [int in 0..59]
```

This is true to start

```
def increment(self, hours, mins):
    """Move this time <hours> hours
    and <mins> minutes into the future.
    Pre: hours [int] >= 0; mins in 0..59"""
```

What we are supposed to accomplish

This is also true to start

```
    self.min  = self.min + mins
    self.hour = self.hour + hours
```

?

You put code here

```
Instance variables:
    hour: hour of day      [int in 0..23]
    min:  minute of hour   [int in 0..59]
```

This should be true at the end

Implementing a Method

```
Instance variables:
  hour: hour of day       [int in 0..23]
  min:  minute of hour [int in 0..59]
```

This is true to start

```
def increment(self, hours, mins):
      """Move this time <hours> hours
      and <mins> minutes into the future.
      Pre: hours [int] >= 0; mins in 0..59"""
```

What we are supposed to accomplish

This is also true to start

```
      self.min  = self.min + mins
      self.hour = (self.hour + hours +
                        self.min / 60)
      self.min  = self.min % 60
      self.hour = self.hour % 24
```

You put code here

```
Instance variables:
  hour: hour of day       [int in 0..23]
  min:  minute of hour [int in 0..59]
```

This should be true at the end

Role of Invariants and Preconditions

- They both serve two purposes
  - Help you think through your plans in a disciplined way
  - Communicate to the user* how they are allowed to use the class
- Provide the *interface* of the class
  - interface btw two programmers
  - interface btw parts of an app
- Important concept for making large software systems
  - Will return to this idea later

*** …who might well be you!**

in•ter•face |ˈintərˌfās| noun

1. a point where two systems, subjects, organizations, etc., meet and interact : the interface between accountancy and the law.
   - *chiefly Physics* a surface forming a common boundary between two portions of matter or space, e.g., between two immiscible liquids : the surface tension of a liquid at its air/liquid interface.
2. *Computing* a device or program enabling a user to communicate with a computer.
   - a device or program for connecting two items of hardware or software so that they can be operated jointly or communicate with each other.

—The Oxford American Dictionary

Implementing a Class

- All that remains is to fill in the methods. (All?!)
- When implementing methods:
  1. Assume precondit
  2. Assume class inv
  3. Ensure method sp
  4. Ensure class invariant when done

Easy(ish) if we are the user.

But what if we aren't?

- Later, when using the class:
  - When calling methods, ensure preconditions are true
  - If attributes are altered, ensure class invariant is true

Recall: Enforce Preconditions with assert

```
def anglicize(n):
    """Returns: the anglicization of int n.

    Precondition: n an int, 0 < n < 1,000,000"""
    assert type(n) == int, str(n)+' is not an int'
    assert 0 < n and n < 1000000, str(n)+' is out of range'
    # Implement method here...
```

Check (part of) the precondition

(Optional) Error message when precondition violated

Enforce Method Preconditions with **assert**

```
class Time(object):
    """Instances represent times of day."""

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"""
        assert type(hour) == int
        assert 0 <= hour and hour < 24
        assert type(min) == int
        assert 0 <= min and min < 60

    def increment(self, hours, mins):
        """Move this time <hours> hours
        and <mins> minutes into the future.
        Pre: hours is int >= 0; mins in 0..59"""
        assert type(hour) == int
        assert type (min) == int
        assert hour >= 0 and
        assert 0 <= min and min < 60
```

```
Instance Attributes:
  hour: hour of day [int in 0..23]
  min:  minute of hour [int in 0..59]
```

Initializer creates/initializes all of the instance attributes.

Asserts in initializer guarantee the initial values satisfy the invariant.

Asserts in other methods enforce the method preconditions.

Hiding Methods From Access

- Put underscore in front of a method will make it **hidden**
  - Will not show up in help()
  - But it is still there…
- Hidden methods
  - Can be used as **helpers** inside of the same class
  - But it is bad style to use them outside of this class
- Can do same for attributes
  - Underscore makes it hidden
  - Do not use outside of class

```
class Fraction(object):
    """Instance attributes:
        numerator:   top       [int]
        denominator: bottom [int > 0]"""

    def _is_denominator(self,d):
        """Return: True if d valid denom"""
        return type(d) == int and d > 0

    def __init__(self,n=0,d=1):
        assert self._is_denominator(d)
        self.numerator = n
        self.denominator = d
```

HIDDEN

Helper method

Enforcing Invariants

```
class Fraction(object):
    """Instance attributes:
        numerator:    top      [int]
        denominator:  bottom   [int > 0]
    """
```

Invariants: Properties that are always true.

- These are just comments!

```
>>> p = Fraction(1,2)
>>> p.numerator = 'Hello'
```

- How do we prevent this?

- **Idea**: Restrict direct access
  - Only access via methods
  - Use asserts to enforce them
- Examples:

```
def getNumerator(self):
    """Returns: numerator"""
    return self.numerator

def setNumerator(self,value):
    """Sets numerator to value"""
    assert type(value) == int
    self.numerator = value
```

Data Encapsulation

- **Idea**: Force the user to only use methods
- Do not allow direct access of attributes

Setter Method

- Used to change an attribute
- Replaces all assignment statements to the attribute
- **Bad**:

```
>>> f.numerator = 5
```

- **Good**:

```
>>> f.setNumerator(5)
```

Getter Method

- Used to access an attribute
- Replaces all usage of attribute in an expression
- **Bad**:

```
>>> x = 3*f.numerator
```

- **Good**:

```
>>> x = 3*f.getNumerator()
```

Data Encapsulation

```
class Fraction(object):
    """Instance attributes:
        _numerator:   top       [int]
        _denominator: bottom [int > 0]"""

    def getDenomenator(self):
        """Returns: numerator attribute"""
        return self._denomenator

    def setDenomenator(self, d):
        """Alters denomenator to be d
        Pre: d is an int > 0"""
        assert type(d) == int
        assert 0 < d
        self._denominator = d
```

Do this for all of your attributes

Getter

Setter

Naming Convention

The underscore means "should not access the attribute directly."

Precondition is same as attribute invariant.

Mutable vs. Immutable Attributes

Mutable

- Can change value directly
  - If class invariant met
  - **Example**: t.color
- Has both getters and setters
  - Setters allow you to change
  - Enforce invariants w/ asserts

Immutable

- Can't change value directly
  - May change "behind scenes"
  - **Example**: t.x
- Has only a getter
  - No setter means no change
  - Getter allows limited access

May ask you to differetiate on the exam

Structure of a Proper Python Class

```python
class Fraction(object):
    """Instances represent a Fraction
     Attributes:
        _numerator: [int]
       _denominator: [int > 0]"""

    def getNumerator(self):
        """Returns: Numerator of Fraction"""
    ...
    def __init__(self,n=0,d=1):
        """Initializer: makes a Fraction"""
     ...
    def __add__(self,q):
        """Returns: Sum of self, q"""
    ...
    def normalize(self):
        """Puts Fraction in reduced form"""
    ...
```

Docstring describing class
Attributes are all **hidden**

Getters and Setters.

Initializer for the class.
Defaults for parameters.

Python operator overloading

Normal method definitions

Exercise: Design a (2D) Circle

- What are the **attributes**?
  - What is the bare minimum we need?
  - What are some extras we might want?
  - What are the invariants?
- What are the **methods**?
  - With just the one circle?
  - With more than one circle?

Advanced Topic Warning!

The following will not be on the exam

If you ask "Will this be on the Exam"

we will be 

Properties: Invisible Setters and Getters

```
class Fraction(object):
    """Instance attributes:
        _numerator:    [int]
        _denominator: [int > 0]"""
    @property
    def numerator(self):
        """Numerator value of Fraction
        Invariant: must be an int"""
        return self._numerator

    @numerator.setter
    def numerator(self,value):
        assert type(value) == int
        self._numerator = value
```

```
>>> p = Fraction(1,2)
>>> x = p.numerator
```

Python converts to

```
>>> x = p.numerator()
```

```
>>> p.numerator = 2
```

Python converts to

```
>>> p.numerator(2)
```

Properties: Invisible Setters and Getters

```
class Fraction(object):
    """Instance attributes:
        _numerator:   [int]
        _denominator: [int > 0]"""
    @property
    def numerator(self):
        """Numerator value of Fraction
        Invariant: must be an int"""
        return self._numerator
    @numerator.setter
    def numerator(self,value):
        assert type(value) == int
        self._numerator = value
```

Specifies that next method is the **getter** for property of the same name as the method

Docstring describing property

Property uses **hidden** attribute.

Specifies that next method is the **setter** for property whose name is numerator.

Properties: Invisible Setters and Getters

```
class Fraction(object):
    """Instance attributes:
        _numerator:    [int]
        _denominator: [int > 0]"""
    @property
    def numerator(self):
        """Numerator value of Fraction
        Invariant: must be an int"""
        return self._numerator

    @numerator.setter
    def numerator(self,value):
        assert type(value) == int
        self._numerator = value
```

Goal: Data Encapsulation
Protecting your data from other, “clumsy” users.

Only the **getter** is required!

If no **setter**, then the attribute is “immutable”.

Replace **Attributes** w/ **Properties**
(Users cannot tell difference)